CS3025 Compiladores

Laboratorio 3.1 – 28 Agosto 2023

La semana pasada probamos flex con un ejemplo sencillo, `líneas.lex`. En este laboratorio continuaremos explorando flex

1. Comandos básicos

Repitiendo que lo que se hizo el laboratorio pasado, generen el código del scanner definido por `líneas.lex` utilizando el comando flex:

```
>> flex lineas.lex
```

El comando genera el archivo en C `lex.yy.c` con (casi) todo el código necesario para la implementación del scanner definido en nuestro archivo lex. Compilen el archivo en C:

```
>> gcc lex.yy.c -lfl     o      >> gcc lex.yy.c -ll
```

¿Que necesitamos hacer si queremos incluir C++ en nuestro scanner? Debemos compilar el archivo generado por flex con un compilador de C++ e.g. g++.
Para esto, copien primero el fichero original a un nuevo fichero, `líneas_cpp.lex`, y cambien las instrucciones de output a instrucciones en C++:

```
%{
#include <iostream>
using namespace std;
int num_lines = 0, num_chars = 0;
%}
%%
\n      { ++num_lines; ++num_chars; }
.       ++num_chars;
%%
int main() {
  yylex();
  cout <<  "# of lines = " << num_lines;
  cout << " # of chars = " << num_chars << endl;
}
```

En Flex, es posible especificar el nombre del archivo de salida con la opción -o. En lugar de `lex.yy.c`, usemos un nombre de fichero con extensión `.cpp` :

```
>> flex -o líneas.lex.cpp líneas_cpp.lex
```

Flex genera un nuevo archivo: `líneas.lex.cpp`. Compilen el nuevo archivo a un ejecutable de nombre `líneas`:

```
>> gcc -o líneas lineas.lex.cpp -lfl
```

El nuevo programa debería compilar y ejecutar sin problemas

2. Definiciones

La primera sección del archivo flex generalmente es utilizada para introducir definiciones de la forma `nombre patrón`. La siguiente especificación flex define expresiones regulares para `digito` y `numero`. La definición para `numero` es luego usada en la sección de reglas junto con un par de reglas más que reconocen al carácter `a` y a cualquier otro carácter que no sea `a`. Crear el archivo `ejemplo.lex` con lo siguiente:

```
%{
#include <iostream>
using namespace std;
%}
digito [0-9]
numero {digito}+
%%
{numero}  cout << "numero " << yytext << endl;
\n        cout << "new line" << endl;
[^a]      cout << "char not a: " << yytext << endl;
a         cout << "Encontramos a" << endl;

%%
int main() {
  yylex();
  cout << "FIN" << endl;
}
```

Ejecutar flex y compilar:

```
>> flex -o ejemplo.lex.cpp ejemplo.lex
>> g++ -o ejemplo ejemplo.lex.cpp
```

¿ Que pasa al ejecutar ejemplo ?

3. Expresiones aritmeticas

Estamos listos para crear una especificación en flex para las operaciones aritméticas del laboratorio de la primera semana. Crear el archivo `exp.lex` con las siguientes definiciones y reglas:

```
%{
#include <iostream>
 using namespace std;
%}
ws [ \t]+
digit    [0-9]
number   {digit}+
%%
{ws} { }
"("          cout << "LPAREN" << endl;
")"          cout << "RPAREN" << endl;
"+"          cout << "PLUS" << endl;
"-"          cout << "MINUS" << endl;
"*"          cout << "MULT" << endl;
"/"          cout << "DIV" << endl;
{number}   cout  << "number: " << yytext << endl;
.          cout << "error:" << yytext << endl;

%%
int main() {
  yylex();
  cout << "Fin" << endl;
}
```

Compilar y ejecutar:

```
>> flex -o exp.lex.cpp exp.lex
>> g++ -o exp exp.lex.cpp
```

Funciona?

4. Agregando funcionalidad extra

Agregar código para sumar todos los números encontrados. Imprimir el resultado al lado de Fin.